Remarks

Introductory Comments

Prior to this amendment, the present application included claims 67-72, 78-88, 90-92, 94-96 and 99-118. With this amendment, Applicant amends the claims as reflected in the Listing of the Claims, cancels claims 96, 99, 100, 109, 110 and 115 without prejudice or disclaimer of the subject matter thereof, and adds new claim 119. Thus, with this Amendment, claims 67-72, 78-88, 90-92, 94, 95, 101-108, 111-114 and 116-119 are pending in the present application.

Telephonic Interview

Applicants thank the Examiner for the courtesies extended during the telephonic interview of October 27, 2006. The rejections under 35 USC § 112, first paragraph, regarding lack of written description were discussed, and the Examiner recommended identifying further areas in the specification regarding each portion of the claims. The prior art rejections regarding U.S. Patent No. 5,748,741 to Johnson and the Collberg article ("A Taxonomy of Obfuscation Transformations") were discussed with reference to claims 78, 106, 67 and amended claim 114, and the Examiner requested that the distinguishing features discussed during the interview be included in this response. The following remarks are intended to address the Examiner's recommendations and requests.

During the interview, the Examiner requested that Applicants identify some places in the Specification where features of "silent guards" are presented. Numerous embodiments of silent guards are described in the specification and examples are given of several embodiments. Some of the discussion concerning silent guard embodiments and their many features are found in the specification on page 28, line 4 – page 34, line 10; page 77, line 1 – page 79, line 23 and page 84, line 22 – page 88, line 6. These references are given by way of example and are not intended to limit the interpretation of the claims. It will be apparent to one skilled in the art, from the description provided throughout the specification, that many other embodiments of silent guards are possible.

<u>Claim Objections</u>

On page 24 of the Office Action, the Examiner indicated that claims 94 and 95 "would be allowable if rewritten in independent form including all of the limitations of the base claim and any intervening claims." Claims 94 and 95 have been rewritten in independent form. Claims 104 and 105 depend on claims 94 and 95, respectively. Accordingly, Applicants respectfully request that claims 94, 95, 104 and 105 be found allowable.

<u>Claim Rejections – 35 USC § 112, first paragraph</u>

Claims 67-72 and 78-87 were rejected under 35 USC § 112, first paragraph, as failing to comply with the written description requirement.

The MPEP states that: "An objective standard for determining compliance with the written description requirement is, "does the description clearly allow persons of ordinary skill in the art to recognize that he or she invented what is claimed." (MPEP § 2163.02, first paragraph, quoting *In re Gosteli*, 872 F.2d 1008, 1012; 10 USPQ2d 1614, 1618 (Fed. Cir. 1989)). "The subject matter of a claim need not be described literally . . . in order for the disclosure to satisfy the description requirement." (MPEP § 2163.02, third paragraph). "To comply with the written description requirement of 35 U.S.C. 112, para. 1 . . . each claim limitation must be expressly, implicitly, or inherently supported in the originally filed disclosure." (MPEP § 2163.05).

"The analysis of whether the specification complies with the written description requirement . . . is conducted from the standpoint of one of skill in the art at the time the application was filed."

(MPEP § 2163.II.A.2) "The examiner has the initial burden of presenting evidence or reasoning to explain why persons skilled in the art would not recognize in the original disclosure a description of the invention defined by the claims." (MPEP § 2163.II.A.3(b)). "The fundamental factual inquiry is whether the specification conveys with reasonable clarity to those skilled in the art that, as of the filing date sought, applicant was in possession of the invention as now claimed." (MPEP § 2163.I.B, last paragraph)

The subject matter covered in claims 67-72 and 78-87 is covered in several places throughout the specification, including page 28, line 4 – page 34, line 10; page 77, line 1 – page 79, line 23 and page 84, line 22 – page 88, line 6. Specific references to some of the occurrences of description in the specification supporting the subject matter of independent claims 67, 78 and 82 is detailed below. These references are given by way of example and are not intended to limit the interpretation of the claims. In fact, it will be appreciated by one of ordinary skill in the art that there are many other ways of implementing the steps recited in the claims.

Claim 67

Claim 67 is directed to a "computer implemented method for adding tamper resistance to a software program." Adding tamper resistance to a software program is discussed throughout the specification, for example the Summary section on pages 5-18, discloses adding tamper resistance on page 6, lines 22-23; page 7, lines 9-10; page 8, lines 1-2; page 12, lines 15-16; pages 13, lines 1-2 and 14-15; page 16, lines 3-4, 10-11 and 19; page 17, lines 4 and 13; and page 18, line 10.

In the Description section, starting on page 23, there are several places where adding or inserting tamper resistance in a software program are disclosed with regard to silent guards. One place is found on page 29, line 20-page 30, line 4, which reads:

In an embodiment of a silent guard according to the present invention, … one or more mathematical expressions comprising the comparison of the run-time value of the variable selected for silent guarding against its expected value are inserted into one or more program instructions from the application software program such that correct execution of the one or more application software program instructions depends upon the run-time value of the variable selected for silent guarding being the same as its expected value.

Another example is found on page 30, lines 5-15 which read:

In an embodiment, the comparison using the run-time value of the variable selected for silent guarding may be indirect. … As before, the comparison may be embodied in one or more mathematical expressions inserted into one or more program instructions from the application software program, so that correct execution of the one or more application software program instructions will depend upon the run-time value of the variable selected for silent guarding being the same as its expected value.

Yet another example is found in the example of silent guarding which begins on page 84 of the specification where on page 85, lines 16-20 read:

This equation then may be adapted for the present example by inserting the following program instructions into the application software program:

$$t = \sqrt{p}$$

$$v = (t + 1)^2 - t^2 - 18$$

$$x = x * v$$

Description supporting the step of "adding a silent guard variable to the software program" can be found on page 28, lines 17-19 which read "the variable selected for silent guarding may be a specialized variable installed in the application software program for the purpose of the silent guard;"

and also in the example on page 31, lines 18-19 which read "the silent guard comprises a variable X . . . Also shown is a variable X1 whose value is computed during the computation of variable X . . . This example also comprises a flag variable F." In the example, silent guard variables X, X1 and F are added to the software program.

Description supporting the step of "selecting a computation in the software program" can be found in the example on page 32 where line 8 has the comment "sample computation of V" and on page 31, lines 16-17 which read "In the pseudocode segment of the following example, variable V is a program variable used in the application software program." In the example, the computation of variable V in the software program is selected for silent guarding. This is also further supported by the description referenced below in support of the following steps.

Description supporting the step of "determining an expected value of the silent guard variable at the execution point of the selected computation" can be found at page 29, lines 3-5 which read "the variable selected for silent guarding has an expected value, i.e., the value the variable should have at a particular point in program execution" and in the example on page 31, lines 18-22 which read:

> the silent guard comprises a variable X whose expected value is 'TRUE.' Also shown is a second variable X1 whose value is computed during the computation of variable X. Variable X1 has an expected value T1. ... The value of flag variable F is initialized to 0, and is changed to 1 after the silent guard program instructions are executed.

In the example, each of the silent guard variables has an expected value, the expected value of the silent guard variable X1 is determined to be T1 at the execution point of the selected computation.

Description supporting the step of "setting the runtime value of the silent guard variable to the expected value of the silent guard variable in the software program at a silent guard insertion point, the silent guard insertion point being separated from the execution point of the selected computation by a plurality of program instructions of the software program" can be found in the example on page 32. In this example, the silent guard variable X1 has an expected value of T1 (see page 31, line 20) and the value of X1 is computed in line 2 of the pseudocode example on page 32 which reads: "2 Compute X1". Thus, in this example, the runtime value of the silent guard variable X1 is set to its expected value at line 2, a silent guard insertion point. Also, in this example, as described above, the selected computation is the computation of the variable V which begins on line 8 of the pseudocode example on page 32. Thus, the silent guard insertion point (line 2 of the pseudocode example) is separated from the execution point of the selected computation (starting on line 8 of the pseudocode example) by a plurality of program instructions of the software program. In addition, the specification on page 32, lines 2-9, which read:

> It will be appreciated by those of skill in the art that other program instructions may be interspersed between pseudocode program instructions shown below to be adjacent without affecting the output of the pseudocode segment. For example, the pseudocode program instruction of line 7 below (identified by label "L") need not be located immediately after the pseudocode program instruction of line 6. Likewise, any of the other pseudocode program instructions shown below to be adjacent may, when implemented as program instructions in a programming language, be separated by one or more other program instructions

This further supports the description of the silent guard insertion point being separated from the execution point of the selected computation by a plurality of program instructions. This step is also further supported by the description referenced below in support of the following steps.

Description supporting the step of "revising the selected computation to be dependent on the runtime value of the silent guard variable, such that the software program executes improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable" can be found on page 33, line 21 – page 34, line 10, which reads:

> The third protection contained in the foregoing pseudocode example arises from the fact that the value of program variable V computed at line 10 will be correct only if the value of Y1 is 0. Thus, the pseudocode program instruction "Y1=X1-T1" shown at line 9 comprises a conditional computation according to the present invention. <u>The value of Y1 will be computed correctly only if the value of variable X1 computed at line 2 is the same as its expected value T1. Accordingly, if the client code block of variable X1 is altered unexpectedly, such as by the actions of a hacker, the value of computed variable X1 will not be the same as its expected value T1. If $X1 \neq T1$, the value of variable Y1 computed at line 9 will be inaccurate and, thus, program variable V will be inaccurately computed at line 10.</u> As before, depending on where and how program variable V is used in the application software program, the inaccurately computed value of program variable V may not become manifest until later in program execution. <u>A hacker altering the computation of variable X1 may have difficulty connecting the altered computation of variable X1 with the failure caused by the inaccurately computed program variable V.</u> (emphasis added)

In addition, the example on page 32 shows the selected computation of program variable V revised to be dependent on the runtime value of the silent guard variable X1, such that the computation of V will evaluate improperly if the runtime value of the silent guard variable X1 is not equal to the expected value T1 of the silent guard variable X1.

Therefore, Applicants submit that claim 67 is described in the Specification along with examples of the steps of claim 67. Claims 68-72 are dependent on claim 67 and are also described in the specification. Accordingly, Applicants respectfully request that the rejection of claims 67-72 under 35 USC § 112, first paragraph, as failing to comply with the written description requirement be withdrawn.

Claim 78

Claim 78 is directed to a "computer implemented method for adding tamper resistance to a software program." As detailed above with regard to claim 67, adding tamper resistance to a software program is discussed throughout the specification. One example is found in the discussion of silent guarding which begins on page 84 of the specification where, on page 85, lines 16-20 read:

> This equation then may be adapted for the present example by inserting the following program instructions into the application software program:
>
> $t = \sqrt{p}$
>
> $v = (t + 1)^2 - t^2 - 18$
>
> $x = x * v$

Description supporting the step of "selecting a program variable in the software program" can be found on page 28, lines 16-17 which read "Typically, the variable selected for silent guarding is a program variable used by the application software program to carry out its intended function;" and lines 21-23 which read "For example, a variable whose value comprises a password entered by a user of the application software program may be selected for silent guarding." This is also shown in the examples on pages 32, 78 and 80 of the specification, in which the variable "V" (pseudocode lines 8 and 10; lines 8 and 10; and lines 10 and 12, respectively) "is a program variable used in the application software program." (page 31, line 17; see also page 79, lines 18-23). This is also shown in the example on page 85 of the specification, in which the variable "*p*" is a password selected for silent guarding (page 85, lines 1-2).

The step of "selecting a computation in the software program" is shown in the example on pages 32, 78 and 80 where pseudocode lines 8, 8 and 10, respectively. have the comment "sample computation of V". Also, page 31, lines 16-17 read "In the pseudocode segment of the following example, variable V is a program variable used in the application software program." In the example, the computation of variable V in the software program is selected for silent guarding. In the example on page 85, the computation of variable *x*, "where *x* is a program variable used in the application software program" (page 85, line 21) is selected. The statement that "*x* is a program variable used in the application software program" would imply to one skilled in the art that *x* is used in "a computation in the software program," especially in view of the examples on pages 32, 78 and 80 with the accompanying descriptions. This is also further supported by the description cited below supporting the following steps.

Description supporting the step of "determining an expected value of the program variable at the point of execution of the selected computation" is shown in the example on page 85 which states "password *p* whose value should be 81" (page 85, line 1). The expected value of program variable *p* is 81. This is also further described on page 29, lines 3-5 which state: "the variable selected for silent guarding has an expected value, i.e., the value the variable should have at a particular point in program execution."

Description supporting the step of "revising the selected computation to be dependent on the runtime value of the program variable, such that the software program executes incorrectly if the runtime value of the program variable is not equal to the expected value of the program variable" is also shown in the example on page 85 which on lines 18-21 shows the sequence of steps:

$$t = \sqrt{p}$$

$$v = (t + 1)^2 - t^2 - 18$$

$$x = x * v$$

and the following explanation "if $p = 81$, then $v = 1$ and the value of program variable x is computed correctly. However, if $p \neq 81$, then $v \neq 1$. If $v \neq 1$, the value of program variable x is computed incorrectly, and the program is corrupted." (page 85, line 21-page 86, line 2). Thus, the computation of x and any calculation in the application program using x, has been revised to be dependent on the runtime value of the program variable p, such that the computation using x will evaluate incorrectly, corrupting the program, if the runtime value of the program variable p is not equal to the expected value of the program variable p.

Therefore, Applicants submit that claim 78 is described in the Specification along with examples of the steps of claim 78. Claims 79-81 are dependent on claim 78 and are also described in the specification. Accordingly, Applicants respectfully request that the rejection of claims 78-81 under 35 USC § 112, first paragraph, as failing to comply with the written description requirement be withdrawn.

Claim 82

Claim 82 is directed to a computer implemented method for adding tamper resistance to a software program. As detailed above with regard to claim 67, adding tamper resistance to a software program is discussed throughout the specification. One example is found in the discussion of silent guarding on page 30, lines 5-15 which read:

In an embodiment, the comparison using the run-time value of the variable selected for silent guarding may be indirect. ... As before, the comparison may be embodied in one or more mathematical expressions inserted into one or more program instructions from the application software program, so that correct execution of the one or more application software program instructions will depend upon the run-time value of the variable selected for silent guarding being the same as its expected value.

Support for the step of "selecting a program block that computes a result necessary for proper execution of the software program, the program block comprising at least one program instruction" is shown on page 32, lines 18-20 which show pseudocode lines:

```
8       V = Z * (A*Z + B) + 72          //sample computation of V
9       Y1 = X1-T1
10      V = V+Y1
```

The program variable V at program line 8 is a computed result necessary for proper execution of the software program (page 34, lines 8-10).

The step of "selecting a silent guard for the program block" is shown by selecting the variable X1 (page 31, lines 15-20). The step of "determining the expected value of the silent guard at the start of execution of the program block" is shown by the variable X1 having an expected value of T1 (page 31, lines 19-20 and page 34, lines 4-5).

The step of "installing a first computation dependent on the silent guard in the software program, such that if the runtime value of the silent guard is not equal to the expected value of the silent guard then the first computation causes the result computed by the program block to evaluate improperly, causing the software program to execute improperly" is shown by pseudocode code lines 9 and 10 on page 32.

Each of pseudocode lines 9 and 10 are a "computation dependent on the silent guard in the software program" where the silent guard is X1. Pseudocode line 9 is explicitly dependent on X1, and pseudocode line 10 is implicitly dependent on X1 because of the dependence of Y1 on the value of X1.

The limitation of claim 82 that "if the runtime value of the silent guard is not equal to the expected value of the silent guard then the first computation causes the result computed by the program block to evaluate improperly, causing the software program to execute improperly" is described for this example on page 34, lines 1-10 which state:

> The value of Y1 will be computed correctly only if the value of variable X1 computed at line 2 is the same as its expected value T1. ... If X1 ≠ T1, the value of variable Y1 computed at line 9 will be inaccurate and, thus, program variable V will be inaccurately computed at line 10. As before, depending on where and how program variable V is used in the application software program, the inaccurately computed value of program variable V may not become manifest until later in program execution. A hacker altering the computation of variable X1 may have difficulty connecting the altered computation of variable X1 with the failure caused by the inaccurately computed program variable V.

Therefore, since claim 82 is described in the Specification along with an example of its steps, Applicants respectfully request that this rejection of claim 82 be withdrawn. Claims 83-87 are dependent on claim 82 and are also described in the specification. Accordingly, Applicants respectfully request that the rejection of claims 82-87 under 35 USC § 112, first paragraph, as failing to comply with the written description requirement be withdrawn.

Claims 70 and 79

Claims 70 and 79 were also rejected under 35 USC § 112, first paragraph, as failing to comply with the written description requirement.

The subject matter covered in claims 70 and 79 is covered in several places throughout the specification, especially in discussions of conditional computations and conditional identities. The following references are given by way of example and are not intended to limit the interpretation of the claims. In fact, it will be appreciated by one of ordinary skill in the art that there are many other ways of implementing the steps recited in the claims.

Claim 70

Claim 70 is dependent on claim 67 and includes some additional steps for revising the selected computation. The example on page 85 which has been discussed above, also exhibits the steps of claim 70. Claim 70 recites "selecting a constant value used in the selected computation." The selected computation in the example on page 85 is a computation using x, "where x is a program variable used in the application software program." (page 85, line 21) It is well known in the art, and in mathematics in general, that any variable z can be replaced by $z*1$ due to the multiplicative identity principle, and the number "1" is a constant. In fact, the expression on page 85, line 20 is effectively $x = x * 1$, when v is equal to its expected value of "1."

The steps on lines 19-20 of page 85 show "replacing the constant value with a mathematical expression that is dependent on the runtime value of the silent guard variable, such that the mathematical expression evaluates to the constant value if the runtime value of the silent guard variable is equal to the expected value of the silent guard variable."

In line 20, the constant value "1" is replaced by the variable v, and line 19 shows the "mathematical expression that is dependent on the runtime value of the silent guard variable," the silent guard variable being t. It would be obvious to those skilled in the art that lines 19 and 20 could be combined by replacing the variable v in the expression on line 20 with the mathematical expression for v shown in line 19, resulting in:

$$x = x * [(t + 1)^2 - t^2 - 18]$$

This is effectively replacing the constant value "1" with "a mathematical expression that is dependent on the runtime value of the silent guard variable" t as recited in claim 70.

As stated on page 85, line 21-page 86, line 2, "Note that if $p = 81$, then $v = 1$ and the value of program variable x is computed correctly. However, if $p \neq 81$, then $v \neq 1$. If $v \neq 1$, the value of program variable x is computed incorrectly, and the program is corrupted." The silent guard variable t which is used in the mathematical expression on line 19, is computed on line 18 as " $t = \sqrt{p}$." The expected value of p is 81 (page 85, line 1), and "the expected value of variable t is computed as $t = \sqrt{81} = 9$" (page 85, line 3). Thus, the statement on page 85, line 21-page 86, line 2 could be rephrased as "if $t = 9$, then $v = 1$ and the value of program variable x is computed correctly. However, if $t \neq 9$, then $v \neq 1$ and the value of program variable x is computed incorrectly, and the program is corrupted." Therefore, as recited in claim 70, "the mathematical expression evaluates to the constant value [1] if the runtime value of the silent guard variable [t] is equal to the expected value [9] of the silent guard variable."

Therefore, Applicants submit that claim 70 is described in the Specification along with at least one example of its implementation. Accordingly, Applicants respectfully request that the rejection of claims 70 under 35 USC § 112, first paragraph, as failing to comply with the written description requirement be withdrawn.

Claim 79

Claim 79 is dependent on claim 78 and includes some additional steps for revising the selected computation. The additional steps of claim 79 are similar to the additional steps of claim 70 except that the mathematical expression of claim 79 is dependent on a program variable, whereas the mathematical expression of claim 70 is dependent on a silent guard variable. The example on page 85 also exhibits the additional steps of claim 79.

Claim 79 recites "selecting a constant value used in the selected computation." The selected computation in the example on page 85 is a computation using x, "where x is a program variable used in the application software program." (page 85, line 21) It is well known in the art, and in mathematics in general, that any variable z can be replaced by z*1 due to the multiplicative identity principle, and the number "1" is a constant. In fact, the expression on page 85, line 20 is effectively $x = x * 1$, when v is equal to its expected value of "1."

The steps on lines 19-20 of page 85 show "replacing the constant value with a mathematical expression that is dependent on the runtime value of the program variable, such that the mathematical expression evaluates to the constant value if the runtime value of the program variable is equal to the expected value of the program variable."

In line 20, the constant value "1" is replaced by the variable v, and lines 18 and 19 show the "mathematical expression that is dependent on the runtime value of the program variable," the program variable being p. It would be obvious to those skilled in the art that lines 18, 19 and 20 could be combined by replacing the variable t in the expression on line 19 with the $\sqrt{p}$ as defined in line 18, and replacing the variable v in the expression on line 20 with the mathematical expression for v shown in line 19, resulting in:

$$x = x * [(\sqrt{p} + 1)^2 - (\sqrt{p})^2 - 18].$$

The resulting mathematical expression is clearly dependent on the program variable p. This is effectively replacing the constant value "1" with "a mathematical expression that is dependent on the runtime value of the program variable" p as recited in claim 79.

This is also described in the Specification on page 85, line 21-page 86, line 2 which states: "Note that if $p = 81$, then $v = 1$ and the value of program variable x is computed correctly. However, if $p \neq 81$, then $v \neq 1$. If $v \neq 1$, the value of program variable x is computed incorrectly, and the program is corrupted." Thus, as recited in claim 79, "the mathematical expression evaluates to the constant value [1] if the runtime value of the program variable [p] is equal to the expected value [81] of the program variable."

Therefore, Applicants submit that claim 79 is described in the Specification along with at least one example of its implementation. Accordingly, Applicants respectfully request that the rejection of claims 79 under 35 USC § 112, first paragraph, as failing to comply with the written description requirement be withdrawn.

Claim Rejections – 35 USC § 102(a)

Claims 78, 80-87, 91, 92, 96 and 99-118 were rejected under 35 USC § 102(b) as being anticipated by U.S. Patent No. 5,748,741 to Johnson (hereinafter "Johnson").

Claim 78

In the rejection of claim 78, the Examiner cites Johnson col. 12:10-14, which state:

```
320:  Insert checks and traps as needed:
    if cascades computed values different from expected, set TamperFlag and start
        a timer
    if TamperFlag is set and timer has expired, go to trap code.
```

The check for tampering disclosed in Johnson appears to be using several computed values calculated by "checking cascades" and comparing these computed values which are "expected" to be equal. Earlier in the above cited algorithm, Johnson recites the steps of:

> 240: Generate checking cascades …
>
> matched pairs of checking cascades need to have matched input values as well

This step 240 of Johnson sets up the "checking cascades" which output the "cascades computed values" that are checked in step 320 of Johnson cited above. The "cascades computed values" are "expected" to be equal. See also, Johnson col. 6: 4-16 which discusses "generating one or more checking cascades which are each similar to the executable program design … inserting periodic checking codes distributed over the checking cascades, the checking codes monitoring expected output of the checking cascades at predetermined points in the executable program design." These are the only references to "expected" in the Johnson specification. Thus, Applicants understand Johnson to teach generating multiple copies of a computation in "checking cascades" and comparing the outputs of these cascades which are "expected" to be equal.

The expected outputs are compared in an inserted IF statement, and IF the computed values are not equal, as expected, THEN the algorithm sets TamperFlag and starts timer. Johnson teaches inserting an explicit compare statement and then delaying the protective action by using a timer.

In contrast, claim 78 recites:

> selecting a program variable in the software program;
>
> selecting a computation in the software program; . . . and
>
> revising the selected computation to be dependent on the runtime value of the

program variable, such that the software program executes incorrectly if the runtime value of the program variable is not equal to the expected value of the program variable.

As opposed to generating multiple copies of a computation and inserting an explicit compare statement to compare the computed outputs as taught by Johnson, claim 78 recites a selected computation in the software program is revised "to be dependent on the runtime value of the program variable, such that the software program executes incorrectly if the runtime value of the program variable is not equal to the expected value of the program variable." Johnson does not teach selecting and revising a computation, but rather teaches generating multiple copies of a computation in "checking cascades" and adding a new statement (an IF statement) into the program to compare the computed outputs.

There are several disadvantages of the Johnson method. The additional copies of a computation in "checking cascades" create a penalty in storage space and computation time, both of which are very scarce in many applications, for example embedded and real time systems. In addition, to overcome the protection a hacker can: (1) look for IF statements, such as the first that compares the computed and expected outputs or the second that branches to the trap code, or (2) find where execution stopped, the trap code, and then trace the flow backwards to find the branch instructions, the IF statements. However, using the method recited in claim 78 does not require additional copies of a computation, and there is not such an obvious statement for the hacker to look for, such as a branch or IF statement, and there is not a specific area where the program will stop execution. When "the runtime value of the program variable is not equal to the expected value of the program variable," the output of the selected computation is corrupted, as would be any further computations dependent on the output of the selected computation, but the program may complete execution or crash. The hacker has a much harder time trying to detect and overcome the protection method of claim 78.

For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claim 78 to be allowable over Johnson. Claims 80, 81 and 101 are dependent on base claim 78. Accordingly, Applicants respectfully request that the Examiner find claims 78, 80, 81 and 101 allowable.

Claim 80

In addition to the reasons given above with regard to claim 78, Applicant further distinguishes claim 80 over Johnson. In the rejection of claim 80, the Examiner cites Johnson col. 10:42-11:18, much of which is shown in Figure 7 of Johnson. This describes a method of obfuscating a computation "a+b" such that the values of "a" and "b" never appear together (col. 10:33-37). The method disclosed includes separating "a" and "b", intertwining them with a third input "c", computing multiple intermediate results "t, u, v, w", and intertwining those results to compute outputs "x" and "y" which can be combined to find the sum "s" of "a+b." This method of Johnson simply teaches an obfuscation method using additive and commutative properties of addition and multiplication to complicate and obfuscate a computation. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In contrast, claim 80 which depends on claim 78, recites steps of:

> determining an expected value of the program variable at the point of execution of the selected computation; (claim 78)
>
> determining an expected value of the computation variable at the point of execution of the selected computation; (claim 80).

In addition, claim 80 recites dependencies that corrupt computations if the runtime value does not equal the expected value, such as:

> replacing the computation variable with a mathematical expression that is dependent on the runtime value of the program variable, such that <u>the mathematical expression evaluates to the expected value of the computation variable if the runtime value of the program variable is equal to the expected value of the program variable</u>. (claim 80)
>
> revising the selected computation to be dependent on the runtime value of the program variable, such that <u>the software program executes incorrectly if the runtime value of the program variable is not equal to the expected value of the program variable</u>. (claim 78)

The method of claim 80 requires determining expected values and revising computations to be dependent on variables such that they compute incorrectly if the runtime value and expected value are not equal.

Johnson does not teach, disclose or suggest the method of claim 80. The portion of Johnson cited by the Examiner uses no expected values or dependencies on expected values. The limited portion of Johnson that mentions the term "expected" appears to be referring to the fact that multiple copies of a computation in multiple "checking cascades" are "expected" to generate the same outputs. These outputs are compared in explicit IF statements that set flags and cause branching to trap codes, which is not as recited in claim 80. For at least these reasons, in addition to the reasons given above with regard to claim 78, Applicants respectfully request that the Examiner withdraw the rejection and find claim 80 to be allowable over Johnson.

<u>Claim 81</u>

In addition to the reasons given above with regard to claim 78, Applicants further distinguish claim 81 over Johnson. In the rejection of claim 81, the Examiner cites Johnson col. 10:42-58. This is the first portion of the obfuscation method of Johnson discussed above with regard to claim 80. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In contrast, claim 81 which depends on claim 78, recites determining, inserting and creating dependencies on expected values. For example, the method of claim 81 recites: "determining an expected value ..." (claim 78), "inserting a mathematical expression including ... the expected value" (claim 81), and "revising the selected computation ... such that the software program executes incorrectly if the runtime value of the program variable is not equal to the expected value of the program variable" (claim 78).

Johnson does not teach, disclose or suggest the method of claim 81. The portion of Johnson cited by the Examiner does not use expected values or dependencies on expected values. The limited portion of Johnson that mentions the term "expected" appears to be referring to the fact that multiple copies of a computation in multiple "checking cascades" are "expected" to generate the same outputs. These outputs are compared in explicit IF statements that set flags and cause branching to trap codes, which is different from claim 81. For at least these reasons, in addition to the reasons given above with regard to claim 78, Applicants respectfully request that the Examiner withdraw the rejection and find claim 81 to be allowable over Johnson.

Claim 82

In the rejection of claim 82, the Examiner cites Johnson col. 12:10-13, which state:

```
if cascades computed values different from expected, set TamperFlag
        and start a timer
if TamperFlag is set and timer has expired, go to trap code.
```

As discussed above with regard to claim 78, Johnson discloses making multiple copies of a computation in "checking cascades" and compares the outputs which are "expected" to be equal, to check for tampering. The check for tampering disclosed in Johnson uses an inserted IF statement: IF computed values are not equal to expected, THEN set TamperFlag and start timer. Thus, Johnson teaches making multiple copies of a computation, comparing the outputs, inserting an explicit compare statement and branching to trap code if the compare fails.

In contrast, amended claim 82 recites:

> selecting a program block that computes a result necessary for proper execution of the software program, …;
>
> selecting a silent guard …;
>
> determining the expected value of the silent guard …; and
>
> installing a first computation dependent on the silent guard …, such that if the runtime value of the silent guard is not equal to the expected value of the silent guard then the first computation causes the result computed by the program block to evaluate improperly …

Johnson does not teach or disclose "selecting a program block that computes a result necessary for proper execution," "determining the expected value of the silent guard" and "installing a first computation dependent on the silent guard …, such that if the runtime value of the silent guard is not equal to the expected value of the silent guard then the first computation causes the result computed by the program block to evaluate improperly." Johnson does not disclose making a "result necessary for proper execution" and causing "the result computed by the program block to evaluate improperly" using a silent guard and a computation.

The Johnson method also has the disadvantages that it takes additional space for the multiple copies of computations in the "checking cascades," it takes additional time to compute the multiple copies of the computation, and a hacker can: (1) look for IF statements, such as the first that compares the computed and expected outputs or the second that branches to the trap code, or (2) find where execution stopped, the trap code, and then trace the flow backwards to find the branch instructions, the IF statements. However, using the method recited in claim 82 does not take as much additional time or space, and there is not such an obvious statement for the hacker to look for, such as a branch or IF statement, and there is not a specific area where the program will stop execution.

For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claim 82 to be allowable over Johnson. Claims 83-87, 102 and 103 are dependent on base claim 82. Accordingly, Applicants respectfully request that the Examiner find claims 82-87, 102 and 103 allowable over Johnson.

Claim 86

In addition to the reasons given above with regard to claim 82, Applicant further distinguishes claim 86 over Johnson. In the rejection of claim 86, the Examiner cites Johnson col. 10:54-11:4; 11:17. This is the first portion of the obfuscation method of Johnson discussed above with regard to claims 80 and 81. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In contrast, claim 86 recites determining expected values, creating dependencies on expected values, and:

> installing a second computation that includes the runtime value of the program

variable, such that the result of the second computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable at the insertion point; and

setting the silent guard equal to the result of the second computation.

The TamperFlag in Johnson, which the Examiner compares with the silent guard, is not made equal to the result of a mathematical computation, as recited in claim 86. In addition, the section of Johnson cited by the Examiner with regard to claim 86 uses no expected values or dependencies on expected values. Johnson does not teach, disclose or suggest setting a silent guard equal to a mathematical computation and using expected values for adding tamper resistance as recited in claim 86. Claim 87 is dependent on claim 86. For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claims 86 and 87 to be allowable over Johnson.

Claim 106

In the rejection of claim 106, the Examiner cites various portions of Johnson col. 10:16-11:18. This section of Johnson describes the obfuscation method discussed above with regard to claims 80 and 81. This section of Johnson discloses an obfuscation method using additive and commutative properties of addition and multiplication to complicate and obfuscate a computation. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In contrast, part of claim 106 recites:

a silent guard variable having an expected value at a first dependency point in the software program;

a program variable having an expected value at a second dependency point in the software program;

a mathematical computation that includes the runtime value of the silent guard variable and an expected term, the expected term being set based on the expected value of the silent guard variable at the first dependency point.

Not only does claim 106 recite expected values for the silent guard variable and the program variable, but it also recites "a mathematical computation that includes the runtime value of the silent guard variable and an expected term ... set based on the expected value of the silent guard variable at the first dependency point." The Examiner relates "t, u, v and w" of Johnson to the silent guard variable, and "2t+v+w and 2u+v+w" of Johnson to the expected term (see paragraph 29q of the Office Action). However, "2t+v+w and 2u+v+w" are not expected terms set based on expected values, but are rather computations using the runtime values of t, u, v and w. Johnson does not disclose or teach determining an expected value of t, u, v or w, nor does Johnson disclose or teach "a mathematical computation that includes the runtime value of the silent guard variable and an expected term ... set based on the expected value of the silent guard variable" as recited in claim 106. Claims 107 and 108 are dependent on claim 106. For at least these reasons, Applicants respectfully request that the Examiner withdraw this rejection and find claims 106, 107 and 108 to be allowable.

Claim 112

Claim 112 formerly depended on claim 109 and intervening claim 110, both of which are no longer pending in the present application. Claim 112 has been amended to include the limitations of former claims 109 and 110. In the rejection of claim 112, the Examiner cites Johnson col. 10:42-49 and 62; and 11:3 and 18. These are various portions of the obfuscation method of Johnson discussed above with regard to claims 80 and 81. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In addition, with regard to the rejection of claim 109 the Examiner cites Johnson col. 12:11-14 and specifically cites col. 12:11 with regard to the silent guard (see paragraph 32t of the Office Action), apparently comparing the silent guard of claim 109-113 to the TamperFlag of Johnson. Johnson col. 12:11-14 is discussed above with regard to claim 78. This portion of Johnson discloses comparing multiple computed outputs that are expected to be equal, inserting an IF statement to set a flag and a timer if they are not equal, and branching to trap code when the timer expires and the flag is set.

In contrast, claim 112 recites expected values and dependencies thereon:

> wherein the runtime value of the silent guard is made dependent on the runtime value of the program variable using a mathematical computation that includes the runtime value of the program variable, wherein the result of the mathematical computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable at the insertion point.

The TamperFlag in Johnson, which the Examiner compares with the silent guard, is only dependent on the IF statement in Johnson col. 12:12-13. The IF statement of Johnson col. 12:12-13 either sets the flag if the values are different or does not set the flag if the values are not different. An IF statement is not corrupted, but simply takes an alternate branch. In contrast, claim 112 recites "the runtime value of the silent guard is made dependent on the runtime value of the program variable using a mathematical computation ... wherein the result of the mathematical computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable." Johnson does not teach, disclose or suggest a mathematical computation that makes the TamperFlag dependent on the runtime value of the program variable wherein the result of the mathematical computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable. An IF statement is not corrupted, but simply takes an alternate branch. Claims 111 and 113 are dependent on claim 112. For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claims 111, 112 and 113 to be allowable.

Claim 114

In the rejection of claim 114, the Examiner states: "As per claims 114-118, they are the method claims corresponding to claims 109-113, ... claims 114-118 are rejected as being unpatentable as being anticipated by Johnson over for the same reasons set forth in the rejections of claim 109-113." (see paragraph 37 of the Office Action) In the rejection of claim 109, the Examiner cites Johnson col. 12:11-14. Johnson col. 12:11-14 is discussed above with regard to claim 78. This portion of Johnson appears to disclose generating multiple copies of a computation in "checking cascades," the outputs of the multiple copies expected to be equal, and inserting an IF statement to set a flag and a timer if they are not equal, and branching to trap code when the timer expires and the flag is set.

Claim 114 has been amended to include steps of:

> determining a program block in the software program having a program variable with an expected value that causes improper execution of the software program when the program variable is not equal to the expected value of the program variable, ...
>
> inserting a silent guard having an expected value ...
>
> making the program variable dependent on the silent guard using a mathematical function, such that the mathematical function does not compute the expected value of the

program value if the silent guard is not equal to the expected value of the silent guard, which causes the software program to execute improperly.

Johnson does not teach or disclose making program variables dependent on silent guards using a mathematical function, as well as other limitations of amended claim 114. For at least these reasons, Applicants respectfully request that the Examiner withdraw this rejection and find claim 114 to be allowable over Johnson.

<u>*Claim 117*</u>

Claim 117 formerly depended on claim 114 and intervening claim 115. Claim 117 has been amended to include the limitations formerly in claims 114 and 115. The Examiner states: "As per claims 114-118, they are the method claims corresponding to claims 109-113, ... claims 114-118 are rejected as being unpatentable as being anticipated by Johnson over for the same reasons set forth in the rejections of claim 109-113." (see paragraph 37 of the Office Action) Applicants submit that claims 114-118 stand on their own and presents the following reasons for their allowance.

In the rejection of claim 109 the Examiner cites Johnson col. 12:11-14 and specifically cites col. 12:11 with regard to the silent guard (see paragraph 32t of the Office Action), apparently comparing the silent guard to the TamperFlag of Johnson. Johnson col. 12:11-14 is discussed above with regard to claim 78. This portion of Johnson appears to disclose generating multiple copies of a computation in "checking cascades," the outputs of the multiple copies expected to be equal, and inserting an IF statement to set a flag and a timer if they are not equal, and branching to trap code when the timer expires and the flag is set. In the rejection of claim 112, the Examiner cites Johnson col. 10:42-49 and 62; and 11:3 and 18. These are various portions of the obfuscation method of Johnson discussed above with regard to claims 80 and 81. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In contrast, claim 117 recites expected values and making dependencies thereon:

> making the runtime value of the silent guard dependent on the runtime value of the program variable using a mathematical computation that includes the runtime value of the program variable, wherein the result of the mathematical computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable at the insertion point, such that the runtime value of the silent guard equals the expected value of the silent guard if the runtime value of the program variable equals the expected value of the program variable at the insertion point, and the branch instruction causes the program block to be executed if the runtime value of the silent guard is not equal to the expected value of the silent guard.

The TamperFlag in Johnson, which the Examiner compares with the silent guard, is only dependent on the IF statement in Johnson col. 12:12-13. The IF statement of Johnson col. 12:12-13 either sets the flag if the values are different or does not set the flag if the values are not different. An IF statement is not corrupted, but simply takes an alternate branch. In contrast, claim 117 recites "making the runtime value of the silent guard dependent on the runtime value of the program variable using a mathematical computation ... wherein the result of the mathematical computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable." Johnson does not teach, disclose or suggest a mathematical computation that makes the TamperFlag dependent on the runtime value of the program variable wherein the result of the mathematical computation is corrupted if the runtime value of the program variable is not equal to the expected value of the program variable. An IF statement is not corrupted, but simply takes an alternate branch. Claims 116 and 118 are dependent on claim 117. For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claims 116-118 to be allowable over Johnson.

Claim 91

The Examiner states: "As per claims 91 and 92, they are the method corresponding to claims 106-108, ... claims 91 and 92 are rejected as being unpatentable as being anticipated by Johnson over for the same reasons set forth in the rejections of claim 106-108." (see paragraph 38 of the Office Action) Though these claims have similarities, Applicants respectfully submit they have differences. However, to address this rejection, the following remarks will address the rejections set forth by the Examiner with regard to claims 106-108.

In the rejection of claim 91 (using the rejection of claim 106 given in paragraph 29 of the Office Action), the Examiner cites various portions of Johnson col. 10:16-11:18. This section of Johnson describes the obfuscation method discussed above with regard to claims 80 and 81. This section of Johnson discloses an obfuscation method using additive and commutative properties of addition and multiplication to complicate and obfuscate a computation. This method of Johnson does not disclose or teach use of expected values, but rather the insertion of multiple computations producing intermediate results, and intertwining the added computations and intermediate results.

In contrast, claim 91 recites steps of:

> a program variable having an expected value at a first dependency point in the software program;
>
> a silent guard variable having an expected value at the first dependency point;
>
> a mathematical computation that includes the runtime value of the silent guard variable and an expected term, the expected term being set based on the

expected value of the silent guard variable at the first dependency point.

Not only does claim 91 recite expected values for a silent guard variable and a program variable, but it also recites "a mathematical computation that includes the runtime value of the silent guard variable and an expected term ... set based on the expected value of the silent guard variable at the first dependency point." The Examiner relates "t, u, v and w" of Johnson to the silent guard variable, and "2t+v+w and 2u+v+w" of Johnson to the expected term (see paragraph 29q of the Office Action). However, "2t+v+w and 2u+v+w" are not expected terms set based on expected values, but are rather computations using the runtime values of t, u, v and w. Johnson does not disclose or teach determining an expected value of t, u, v or w, nor does Johnson disclose or teach "a mathematical computation that includes the runtime value of the silent guard variable and an expected term ... set based on the expected value of the silent guard variable" as recited in claim 91. Claim 92 is dependent on claim 91. For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claims 91 and 92 to be allowable.

<u>Claim Rejections – 35 USC § 103(a)</u>

Claims 67-72, 79, 88 and 90 were rejected under 35 USC § 103(a) as being unpatentable over the Collberg article entitled "A Taxonomy of Obfuscation Transformations" (hereinafter "Collberg") in view of Johnson.

Collberg identifies 4 classes of technical program protection: *Obfuscation, Encryption, Server side execution* and *Trusted code* (Figure 1(a)), and then focuses almost entirely on obfuscation (Figure 1(b)-(g); Abstract, page 1, col. 1; and Introduction, page 2, col. 1, line 4 - col. 2, line 7). Collberg divides the targets of obfuscating transformations into 4 categories, *Layout obfuscation*, *Data obfuscation*, *Control obfuscation* and *Preventive transformation*

(Figure 1(c)) and then presents a catalogue of obfuscating transformations for each of these categories (Figure 1(d)-(g)).

In the Office Action, the Examiner rejects claims 67-72 and 79 based on Collberg, Section 7.1.3 entitled "Split Variable" (page 18) and the accompanying Figure 18 (page 19). Split variable is a type of data obfuscation transformation (Collberg, page 2, Figure 1(e); page 17, Section 7 entitled "Data Transformations").

Collberg makes clear that the obfuscation techniques described in the paper "converts a program into an equivalent one that is more difficult to understand and reverse engineer." (Collberg, page 1, col. 1, lines 12-14; *see also*; page 3, col. 2, lines 21-25, and page 4, col. 2, lines 17-18).

The data obfuscation transformations described in Section 7.1.3 of Collberg are one example of an obfuscating transformation from a source program P into a target program P' which Collberg defines on page 6, column 1, as:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate;
- Otherwise, P' must terminate and produce the same output as P.

The "Otherwise" portion of this definition can be rephrased in the positive as:

- If P terminates without an error condition, then P' must terminate and produce the same output as P.

This again emphasizes that, if the original program works (i.e., terminates without an error condition) then the transformed program must also work (i.e., terminate and produce the same output as the original program).

Collberg does not disclose, teach or suggest "determining an expected value" and causing a program to execute improperly if the runtime value of a variable is not equal to an expected value for that variable as required by several of the rejected claims. Collberg is always using runtime values. For clarification of this point, the following explanation goes through the transformation steps shown in Figure 18(e) of Collberg.

Step 1-1': transforms A, B, C to a1, a2, b1, b2, c1, c2.

Step 2-2': transforms True to 0, 1 using Table 18(b). This is a straight transformation with no use of expected values.

Step 3-3': transforms False to 0, 0 using Table 18(b). This is a straight transformation with no use of expected values.

Step 4-4': transforms False to 1, 1 using Table 18(b). This is a straight transformation with no use of expected values.

Step 5-5': transforms the "&" function using Table 18(c) and arithmetic operations. Note this transformation works for any values of A, B and C (see truth tables below).

Collberg, Fig. 18(e), line (5)

A	B	C = A & B
TRUE	TRUE	**TRUE**
TRUE	TRUE	**TRUE**
TRUE	FALSE	**FALSE**
TRUE	FALSE	**FALSE**
FALSE	TRUE	**FALSE**
FALSE	TRUE	**FALSE**
FALSE	FALSE	**FALSE**
FALSE	FALSE	**FALSE**

Collberg, Fig. 18(e), line (5')

A	a1	a2	B	b1	b2	2*a1+a2	2*b1+b2	x	c1	c2	C
TRUE	0	1	TRUE	0	1	1	1	1	0	1	**TRUE**
	1	0		1	0	2	2	1	0	1	**TRUE**
TRUE	0	1	FALSE	0	0	1	0	0	0	0	**FALSE**
	1	0		1	1	2	3	0	0	0	**FALSE**
FALSE	0	0	TRUE	0	1	0	1	3	1	1	**FALSE**
	1	1		1	0	3	2	3	1	1	**FALSE**
FALSE	0	0	FALSE	0	0	0	0	3	1	1	**FALSE**
	1	1		1	1	3	3	3	1	1	**FALSE**

Regardless of whether A and B are TRUE or FALSE, the function in line (5') computes the same result as the function in line (5), namely C = A&B. This is a requirement of Collberg, (see pg. 18, Section 7.1.3). This is a straight transformation with no use of expected values.

Step 6-6': transforms the "&" function using logic operations. Note this transformation works for any values of A, B and C (see truth tables below).

Collberg, Fig. 18(e), line (6)

A	B	C = A & B
TRUE	TRUE	**TRUE**
TRUE	TRUE	**TRUE**
TRUE	FALSE	**FALSE**
TRUE	FALSE	**FALSE**
FALSE	TRUE	**FALSE**
FALSE	TRUE	**FALSE**
FALSE	FALSE	**FALSE**
FALSE	FALSE	**FALSE**

Collberg, Fig. 18(e), line (6')

A	a1	a2	B	b1	b2	a1 ^ a2	b1 ^ b2	c1	c2	C
TRUE	0	1	TRUE	0	1	1	1	1	0	**TRUE**
	1	0		1	0	1	1	1	0	**TRUE**
TRUE	0	1	FALSE	0	0	1	0	0	0	**FALSE**
	1	0		1	1	1	0	0	0	**FALSE**
FALSE	0	0	TRUE	0	1	0	1	0	0	**FALSE**
	1	1		1	0	0	1	0	0	**FALSE**
FALSE	0	0	FALSE	0	0	0	0	0	0	**FALSE**
	1	1		1	1	0	0	0	0	**FALSE**

Regardless of whether A and B are TRUE or FALSE, the function in line (6') computes the same result as the function in line (6), namely C = A&B. This is a requirement of Collberg, (see pg. 18, Section 7.1.3). This is a straight transformation with no use of expected values.

Step 7-7': transforms the "|" function using Table 18(d) and arithmetic operations. Note this transformation works for any values of A, B and C (see truth tables below).

Collberg, Fig. 18(e), line (7)

A	B	C = A | B
TRUE	TRUE	**TRUE**
TRUE	TRUE	**TRUE**
TRUE	FALSE	**TRUE**
TRUE	FALSE	**TRUE**
FALSE	TRUE	**TRUE**
FALSE	TRUE	**TRUE**
FALSE	FALSE	**FALSE**
FALSE	FALSE	**FALSE**

Collberg, Fig. 18(e), line (7')

A	a1	a2	B	b1	b2	2*a1+a2	2*b1+b2	x	c1	c2	C
TRUE	0	1	TRUE	0	1	1	1	1	0	1	**TRUE**
	1	0		1	0	2	2	1	0	1	**TRUE**
TRUE	0	1	FALSE	0	0	1	0	1	0	1	**TRUE**
	1	0		1	1	2	3	2	1	0	**TRUE**
FALSE	0	0	TRUE	0	1	0	1	1	0	1	**TRUE**
	1	1		1	0	3	2	1	0	1	**TRUE**
FALSE	0	0	FALSE	0	0	0	0	3	1	1	**FALSE**
	1	1		1	1	3	3	0	0	0	**FALSE**

Regardless of whether A and B are TRUE or FALSE, the function in line (7') computes the same result as the function in line (7), namely C = A|B. This is a requirement of Collberg, (see pg. 18, Section 7.1.3). This is a straight transformation with no use of expected values.

Step 8-8': transforms A using Table 18(b) and arithmetic operations. Note this transformation works for any value of A (see truth tables below).

Collberg, Fig. 18(e):

line (8)

A
TRUE
TRUE
FALSE
FALSE

line (8')

A	a1	a2	x=2*a1+a2	x==1	x==2	((x==1) || (x==2))
TRUE	0	1	1	TRUE	FALSE	**TRUE**
	1	0	2	FALSE	TRUE	**TRUE**
FALSE	0	0	0	FALSE	FALSE	**FALSE**
	1	1	3	FALSE	FALSE	**FALSE**

Regardless of whether A is TRUE or FALSE, the function in line (8') computes the same result for the IF instruction as the function in line (8), namely A. This is a requirement of Collberg, (see pg. 18, Section 7.1.3). This is a straight transformation with no use of expected values.

Step 9-9': transforms B using Table 18(b) and logic operations. Note this transformation works for any value of B (see truth tables below).

Collberg, Fig. 18(e):

line (9)

B
TRUE
TRUE
FALSE
FALSE

line (9')

B	b1	b2	b1 ^ b2
TRUE	0	1	TRUE
	1	0	TRUE
FALSE	0	0	FALSE
	1	1	FALSE

Regardless of whether B is TRUE or FALSE, the function in line (9') computes the same result for the IF instruction as the function in line (9), namely B. This is a requirement of Collberg, (see pg. 18, Section 7.1.3). This is a straight transformation with no use of expected values.

Step 10-10': transforms C using Table 18(b). Note this transformation works for any value of C (see truth tables below).

Collberg, Fig. 18(e):

line (10)

C
TRUE
TRUE
FALSE
FALSE

line (10')

C	c1	c2	VAL (c1,c2)
TRUE	0	1	TRUE
	1	0	TRUE
FALSE	0	0	FALSE
	1	1	FALSE

Regardless of whether C is TRUE or FALSE, the function in line (10') computes the same result for the IF instruction as the function in line (10), namely C. This is a requirement of Collberg, (see pg. 18, Section 7.1.3). This is a straight transformation with no use of expected values.

Neither in Figure 18 nor elsewhere in Collberg, does Collberg disclose, teach or suggest "determining an expected value" and causing a program to execute improperly if the runtime value of a variable is not equal to an expected value for that variable as required by several of the rejected claims. The section of Collberg cited by the Examiner discloses a transformation from one set of variables A, B, C to another a1, a2, b1, b2, c1, c2. There is no disclosure of expected values, and no disclosure of causing a program to execute improperly if a runtime value is not equal to an expected value. Both of these limitations appear in all of claims 67-72, 79, 88 and 90, as described below. For at least these reason, Applicants respectfully request that the Examiner withdraw the rejection with regard to Collberg.

Claim 67

In the rejection of claim 67, for the step of: "determining an expected value of the silent guard variable at the execution point of the selected computation" the Examiner cites Collberg, "pg. 19, fig. 18(e), steps 5', 7' and 8'; assignment of x" (see Office Action, paragraph 40x). Collberg does not disclose expected values of the silent guard variable. The Examiner compares x of Collberg with a silent guard variable. However, x is an interim variable used in runtime calculations with a1, a2, b1, b2, c1, c2. There is no expected value of x, rather x is an intermediate value used in runtime calculations with a1, a2, b1, b2, c1, c2. Accordingly, Applicants submit that claim 67 distinguishes over Collberg.

In addition, Claim 67 recites a step of: "revising the selected computation to be dependent on the runtime value of the silent guard variable, such that the software program executes improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable." For this step, the Examiner cites Collberg, "pg. 19, fig. 18(e), steps 5', 7' and 8'; assignment of c1 and c2" (see Office Action, paragraph 40z). As explained above, Collberg does not disclose use of expected values, nor does it disclose revising a computation "such that the software program executes improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable." Accordingly, Applicants submit that claim 67 distinguishes over Collberg.

The Examiner also cites Johnson col. 12:13-15 in rejecting the step of: "revising the selected computation to be dependent on the runtime value of the silent guard variable, such that the software program executes improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable." (see Office Action, paragraph 42). As explained above with regard to claim 78, Johnson teaches inserting an explicit IF statement. In contrast, claim 67 recites revising a computation in the software program "to be dependent on the runtime value of the silent guard variable, such that the software program executes improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable."

For at least these reason, Applicants respectfully request that the Examiner withdraw the rejection and find claim 67 to be allowable over Collberg in view of Johnson. Claims 68-72 are dependent on base claim 67. Accordingly, Applicants respectfully request that the Examiner find claims 67-72 allowable.

Claim 68

In addition to the reasons given above with regard to claim 67, Applicant further distinguishes claim 68 over Collberg in view of Johnson. In the rejection of claim 68, the Examiner cites Collberg section 7.1.3 and Fig 18(a-e) (Office Action, paragraph 43). In addition to "determining an expected value of the silent guard variable at the execution point of the selected computation" recited in claim 67, claim 68 recites "determining an expected value of the selected program variable at a dependency point in the software program." Collberg does not disclose expected values of the silent guard variable or of program variables. Collberg discloses a transformation that works for all runtime values of the variables. There is no expected value of x, a1, a2, b1, b2, c1, c2. For at least these reasons, in addition to the reasons described with regard to claim 67, Applicants respectfully request that the Examiner withdraw the rejection and find claim 68 to be allowable over Collberg in view of Johnson.

Claim 69

Claim 69 is dependent on claim 68 and base claim 67 and recites the further steps of "computing the runtime value of the silent guard variable using a mathematical expression including the runtime value of the selected program variable and the expected value of the selected program variable at the dependency point." In addition to the reasons given above with regard to claims 67 and 68, Applicant further distinguishes claim 69 over Collberg in view of Johnson. In the rejection of claim 69, the Examiner cites Collberg, Fig. 18(e) steps 5', 7' and 8' (Office Action, paragraph 44). Steps 5', 7' and 8' of Collberg only show the use of runtime values. Collberg does not disclose, teach or suggest "a mathematical expression including the runtime value of the selected program variable and the expected value of the selected program variable" as recited in claim 69.

For at least these reasons, in addition to the reasons given with regard to claims 67 and 68, Applicants respectfully request that the Examiner withdraw the rejection and find claim 69 to be allowable over Collberg in view of Johnson.

Claim 71

Claim 71 is dependent on claim 67 and recites the further steps of:

selecting a computation variable used in the selected computation;

determining an expected value of the computation variable at the execution point of the selected computation; and

replacing the computation variable with a mathematical expression that is dependent on the runtime value of the silent guard variable, such that the mathematical expression evaluates to the expected value of the computation variable if the runtime value of the silent guard variable is equal to the expected value of the silent guard variable.

In addition to the reasons given above with regard to claim 67, Applicant further distinguishes claim 71 over Collberg in view of Johnson. In the rejection of claim 71, the Examiner cites Collberg, Fig. 18(e) steps 5', 7' and 8' (Office Action, paragraph 46). As explained above, Collberg does not disclose the use of expected values as recited in claim 71. Steps 5', 7' and 8' of Collberg only show the use of runtime values. In addition, Collberg does not disclose, teach or suggest "a mathematical expression that ... evaluates to the expected value of the computation variable if the runtime value of the silent guard variable is equal to the expected value of the silent guard variable" as recited in claim 71. Steps 5', 7' and 8' of Collberg do not show the use of expected values, or mathematical expressions that evaluate to expected values.

For at least these reasons, in addition to the reasons given with regard to claim 67, Applicants respectfully request that the Examiner withdraw the rejection and find claim 71 to be allowable over Collberg in view of Johnson.

Claim 72

Claim 72 is dependent on claim 67 and recites the further steps of "inserting a mathematical expression including the runtime value of the silent guard variable and the expected value of the silent guard variable into the selected computation." In addition to the reasons given above with regard to claim 67, Applicant further distinguishes claim 72 over Collberg in view of Johnson. In the rejection of claim 72, the Examiner cites Collberg, Fig. 18(e) steps 5'-10' (Office Action, paragraph 47). Steps 5'-10' of Collberg only show the use of runtime values. Collberg does not disclose, teach or suggest "a mathematical expression including the runtime value of the silent guard variable and the expected value of the silent guard variable" as recited in claim 72.

For at least these reasons, in addition to the reasons given with regard to claim 67, Applicants respectfully request that the Examiner withdraw the rejection and find claim 72 to be allowable over

Collberg in view of Johnson.

Claim 79

In the rejection of claim 79, the Examiner states "the rejection of claim 78 under 35 USC 102(b) as being anticipated by Johnson is incorporated herein." (see paragraph 48 of the Office Action) Claim 79 is dependent on claim 78, and the Examiner relies on the rejection of claim 78as being anticipated by Johnson for this rejection of claim 79. Applicants submit that claim 78 distinguishes over Johnson for at least the reasons given above in the section regarding claim 78 and the rejection under 35 USC 102(b) as being anticipated by Johnson. Since claim 78 is believed to be allowable, and claim 79 is dependent thereon, Applicants submit that claim 79 is allowable. For at least these reasons, Applicants respectfully request that the Examiner withdraw this rejection and find claim 79 to be allowable over Collberg in view of Johnson.

Claim 88

Claim 88 recites expected values for a silent guard variable and a program variable. In addition, claim 88 recites the silent guard being dependent on the program variable "such that the runtime value of the silent guard variable will not equal the expected value of the silent guard variable at the execution point if the runtime value of the program variable is not equal to the expected value of the program variable at the dependency point." Claim 88 also recites the program computation being dependent on the silent guard variable "such that the program computation will evaluate improperly and the software program will execute improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable." Collberg does not disclose expected values, nor does it disclose computations or software programs that execute improperly if the runtime value of a variable is not equal to an expected value. Johnson discloses a comparison of outputs from multiple copies of a computation, which are expected to be equal, in an IF statement that will cause an eventual branch to trap code if the computed and expected outputs are different. Neither Collberg nor Johnson, together or individually, disclose a program computation dependent on a silent guard variable "such that the program computation will evaluate improperly and the software program will execute improperly if the runtime value of the silent guard variable is not equal to the expected value of the silent guard variable" as recited in claim 88.

For at least these reasons, Applicants respectfully request that the Examiner withdraw the rejection and find claim 88 to be allowable. Claim 90 is dependent on claim 88. Accordingly, Applicants respectfully request that the Examiner find claims 88 and 90 to be allowable over Collberg in view of Johnson.

New Claim

Claim 119 has been added and depends on pending claim 114. Applicants submit that the limitations of claim 119 are supported in the specification by, for example and not by way of limitation, pseudocode line 9 on page 32 of the Specification which shows a computation with both the runtime (X1) and expected values (T1) of a silent guard (X1). Applicants respectfully request that claim 119 be considered and be found allowable.

Final Remarks

Claims 67-72, 78-88, 90-92, 94, 95, 101-108, 111-114 and 116-119 are pending in the present application and are believed to be in condition for allowance. Such allowance is respectfully requested.

In the event that there are any questions related to these amendments or to the application in general, the undersigned would appreciate the opportunity to address those questions directly in a telephone interview at 919-861-5092 to expedite the prosecution of this application for all concerned. If necessary, please consider this a Petition for Extension of Time to affect a timely response. Please charge any additional fees or credits to the account of Bose McKinney & Evans, LLP Deposit Account No. 02-3223.

Respectfully submitted,

BOSE McKINNEY & EVANS LLP

____

Anthony P. Filomena
Reg. No. 44,108

798325_1